

**Adapted from: Nios II Custom Instruction**

**User Guide**

**By Mircea Stan**

**Updated 23 September with “Setting Up the System”**

101 Innovation Drive San Jose, CA 95134 [www.altera.com](http://www.altera.com/)

[**Chapter 1. Nios II Custom Instruction** **Overview**](#_bookmark0)

[Custom Instruction Overview 1–2](#_bookmark2)

[Implementing Custom Instruction Hardware 1–2](#_bookmark3)

[Implementing Custom Instruction Software 1–3](#_bookmark5)

[Custom Instruction Types 1–3](#_bookmark6)

[Combinational Custom Instructions 1–4](#_bookmark10)

[Multicycle Custom Instructions 1–5](#_bookmark14)

[Extended Custom Instructions 1–7](#_bookmark18)

[Internal Register File Custom Instructions 1–9](#_bookmark20)

[External Interface Custom Instructions 1–10](#_bookmark23)

[Chapter 2. Software Interface](#_bookmark25)

[Custom Instruction Examples 2–1](#_bookmark27)

[Built-in Functions and User-defined Macros 2–2](#_bookmark30)

[Custom Instruction Assembly Software Interface 2–3](#_bookmark34)

[Chapter 3. Implementing a Nios II Custom Instruction in Qsys](#_bookmark59)

[Design Example: Cyclic Redundancy Check 4–1](#_bookmark61)

[Implementing Custom Instruction Hardware in Qsys 4–1](#_bookmark62)

[Setting up the Design Environment for the Design Example 4–2](#_bookmark63)

[Opening the Component Editor 4–2](#_bookmark64)

[Adding the HDL Files 4–2](#_bookmark65)

[Configuring the Custom Instruction Signal Type 4–3](#_bookmark66)

[Setting Up the Custom Instruction Interfaces 4–3](#_bookmark67)

[Setting the Details 4–4](#_bookmark68)

[Saving and Adding the Custom Instruction 4–5](#_bookmark69)

[Generating the System and Compiling in the Quartus II Software 4–5](#_bookmark70)

[Accessing the Custom Instruction from Software 4–5](#_bookmark71)

[Using the User-defined Custom Instruction Macro 4–8](#_bookmark74)

[**Appendix A. Custom Instruction Templates**](#_bookmark77)

[VHDL Custom Instruction Template . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . A–2](#_bookmark78)

[Verilog HDL Custom Instruction Template Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . A–3](#_bookmark80)

**iv Contents**

[**Appendix B. Custom Instruction Built-in Functions**](#_bookmark82)

[Built-in Functions that Return a Void Value . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . B–1](#_bookmark84)

[Built-in Functions that Return a Value of Type Int . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . B–1](#_bookmark85)

[Built-in Functions that Return a Value of Type Float . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . B–2](#_bookmark86)

[Built-in Functions that Return a Pointer Value . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . B–2](#_bookmark87)

[**Appendix C. Floating Point Custom Instructions**](#_bookmark88)

When you design a system that includes an Altera Nios II embedded processor, you can accelerate time-critical software algorithms by adding custom instructions to the Nios II processor instruction set. Custom instructions allow you to reduce a complex sequence of standard instructions to a single instruction implemented in hardware. You can use this feature for a variety of applications, for example, to optimize software inner loops for digital signal processing (DSP), packet header processing, and computation-intensive applications. In Qsys, each custom instruction is a separate component in the Qsys system. You can add as many as 256 custom instructions to your system.

In Qsys, the custom instruction logic connects directly to the Nios II arithmetic logic unit (ALU) as shown in [Figure 1–1](#_bookmark1).

**Figure 1–1. Custom Instruction Logic Connects to the Nios II ALU in Qsys Systems**

Nios II Embedded Processor

Custom

Logic

A

Nios II ALU

+

-

<<

>>

Result

&

B

This chapter contains the following sections:

* [“Custom Instruction Overview”](#_bookmark2)
* [“Custom Instruction Types” on page 1–3](#_bookmark6)

For information about the custom instruction software interface, refer to [Chapter 2,](#_bookmark26) [Software Interface](#_bookmark26). For step-by-step instructions for implementing a custom instruction, refer to [Chapter 3, Implementing a Nios II Custom Instruction in Qsys](#_bookmark60).

## Custom Instruction Overview

Nios II custom instructions are custom logic blocks adjacent to the ALU in the processor’s datapath. Custom instructions give you the ability to tailor the Nios II processor core to meet the needs of a particular application. You can accelerate time critical software algorithms by converting them to custom hardware logic blocks.

Because it is easy to alter the design of the FPGA-based Nios II processor, custom instructions provide an easy way to experiment with hardware-software tradeoffs at any point in the design process.

### Implementing Custom Instruction Hardware

[Figure 1–2](#_bookmark4) is a hardware block diagram of a Nios II custom instruction.

**Figure 1–2. Hardware Block Diagram of a Nios II Custom Instruction**

Conduit interface to external memory, FIFO, or other logic

dataa[31..0]

datab[31..0]

clk clk\_en reset start

n[7..0]

a[4..0]

readra

b[4..0]

readrb

c[4..0]

writerc

result[31..0]

Combinatorial

|  |  |
| --- | --- |
| Combinational |  |
|  |
| Multi-cycle |
|  |
| Extended |
| Internal Register File |

done

A Nios II custom instruction logic receives input on its dataa port, or on its dataa and datab ports, and drives out the result on its result port. The custom instruction logic provides a result based on the inputs provided by the Nios II processor.

The Nios II processor supports different types of custom instructions. [Figure 1–2](#_bookmark4) lists the additional ports that accommodate different custom instruction types. Only the ports used for the specific custom instruction implementation are required.

[Figure 1–2](#_bookmark4) also shows a conduit interface to external logic. The interface to external logic allows you to include a custom interface to system resources outside of the Nios II processor datapath.

### Implementing Custom Instruction Software

The Nios II custom instruction software interface is simple and abstracts the details of the custom instruction from the software developer. For each custom instruction, the Nios II Embedded Design Suite (EDS) generates a macro in the system header file, **system.h**. You can use the macro directly in your C or C++ application code, and you do not need to program assembly code to access custom instructions. Software can also invoke custom instructions in Nios II processor assembly language.

For more information about the custom instruction software interface, refer to [Chapter 2, Software Interface](#_bookmark26).

## Custom Instruction Types

Different types of custom instructions are available to meet the requirements of your application. The type you choose determines the hardware interface for your custom instruction.

[Table 1–1](#_bookmark8) shows the available custom instruction types, applications, and associated hardware ports.

**Table 1–1. Custom Instruction Types, Applications, and Hardware Ports (Part 1 of 2)**

|  |  |  |
| --- | --- | --- |
| **Type** | **Application** | **Hardware Ports** |
| Combinational | Single clock cycle custom logic blocks. | ■ dataa[31:0]  ■ datab[31:0]   * result[31:0] |
| Multicycle | Multi-clock cycle custom logic blocks of fixed or variable durations. | ■ dataa[31:0]  ■ datab[31:0]   * result[31:0] * clk * clk\_en [*(1)*](#_bookmark9) * start * reset * done |
| Extended | Custom logic blocks that are capable of performing multiple operations | ■ dataa[31:0]  ■ datab[31:0]   * result[31:0] * clk * clk\_en [*(1)*](#_bookmark9) * start * reset * done   ■ n[7:0] |

**Table 1–1. Custom Instruction Types, Applications, and Hardware Ports (Part 2 of 2)**

|  |  |  |
| --- | --- | --- |
| **Type** | **Application** | **Hardware Ports** |
| Internal register file | Custom logic blocks that access internal register files for input or output or both. | ■ dataa[31:0] |
| ■ datab[31:0] |
| * result[31:0] |
| * clk |
| * clk\_en |
| * start |
| * reset |
| * done |
| ■ n[7:0] |
| ■ a[4:0] |
| * readra |
| ■ b[4:0] |
| * readrb |
| ■ c[4:0] |
| * writerc |
| External interface | Custom logic blocks that interface to logic outside of the Nios II processor’s datapath | Standard custom instruction ports, plus user-defined interface to external logic. |

**Note to** [**Table 1–1**](#_bookmark8)**:**

(1) The clk\_en input signal must be connected to the clk\_en signals of all the registers in the custom instruction, in case the Nios II processor needs to stall the custom instruction during execution.

The following sections discuss the basic functionality and hardware interface of each of the custom instruction types listed in [Table 1–1](#_bookmark8).

### Combinational Custom Instructions

A combinational custom instruction is a logic block that completes its logic function in a single clock cycle. [Figure 1–3](#_bookmark11) shows a block diagram of a combinational custom instruction.

**Figure 1–3. Combinational Custom Instruction Block Diagram**

dataa[31..0]

datab[31..0]

Combinational result[31..0]

In [Figure 1–3](#_bookmark11) the dataa and datab ports are inputs to the logic block, which drives the results on the result port. Because the logic function completes in a single clock cycle, a combinational custom instruction does not require control ports.

[Table 1–2](#_bookmark12) describes the combination custom instruction ports.

**Table 1–2. Combinational Custom Instruction Ports**

|  |  |  |  |
| --- | --- | --- | --- |
| **Port Name** | **Direction** | **Required** | **Description** |
| dataa[31:0] | Input | No | Input operand to custom instruction |
| datab[31:0] | Input | No | Input operand to custom instruction |
| result[31:0] | Output | Yes | Result of custom instruction |

The only required port for combinational custom instructions is the result port. The dataa and datab ports are optional. Include them only if the custom instruction functionality requires input operands. If the custom instruction requires only a single input port, use dataa.

[Figure 1–4](#_bookmark13) shows the combinational custom instruction hardware port timing diagram.

In [Figure 1–4](#_bookmark13), the processor presents the input data on the dataa and datab ports on the rising edge of the processor clock. The processor reads the result port on the rising edge of the following processor clock cycle.

**Figure 1–4. Combinational Custom Instruction Timing Diagram**

clk

T0 T1

T2 T3 T4

dataa[ ]

datab[ ]

result[ ]

dataa[ ] valid

datab[ ] valid

result valid

The Nios II processor issues a combinational custom instruction speculatively; that is, it optimizes execution by issuing the instruction before knowing whether it is necessary, and ignores the result if it is not required. Therefore, a combinational custom instruction must not have side effects. In particular, a combinational custom instruction cannot have an external interface.

You can further optimize combinational custom instructions by implementing the extended custom instruction. Refer to [“Extended Custom Instructions” on page 1–7](#_bookmark18).

### Multicycle Custom Instructions

Multicycle or sequential, custom instructions consist of a logic block that requires two or more clock cycles to complete an operation. Additional control ports are required for multicycle custom instructions, as shown in [Table 1–3](#_bookmark16).

[Figure 1–5](#_bookmark15) shows the multicycle custom instruction block diagram.

**Figure 1–5. Multicycle Custom Instruction Block Diagram**

dataa[31..0]

datab[31..0]

clk clk\_en

reset start

result[31..0]

|  |  |
| --- | --- |
| Multi-cycle |  |
|  |
|  |

done

Multicycle custom instructions complete in either a fixed or variable number of clock cycles. For a custom instruction that completes in a fixed number of clock cycles, you specify the required number of clock cycles at system generation. For a custom instruction that requires a variable number of clock cycles, you instantiate the start and done ports. These ports participate in a handshaking scheme to determine when the custom instruction execution is complete.

[Table 1–3](#_bookmark16) describes the multicycle custom instruction ports.

**Table 1–3. Multicycle Custom Instruction Ports**

|  |  |  |  |
| --- | --- | --- | --- |
| **Port Name** | **Direction** | **Required** | **Description** |
| clk | Input | Yes | System clock |
| clk\_en | Input | Yes | Clock enable |
| reset | Input | Yes | Synchronous reset |
| start | Input | No | Commands custom instruction logic to start execution |
| done | Output | No | Custom instruction logic indicates to the processor that execution is complete |
| dataa[31:0] | Input | No | Input operand to custom instruction |
| datab[31:0] | Input | No | Input operand to custom instruction |
| result[31:0] | Output | No | Result of custom instruction |

As indicated in [Table 1–3](#_bookmark16), the clk, clk\_en, and reset ports are required for multicycle custom instructions. However, the start, done, dataa, datab, and result ports are optional. Implement them only if the custom instruction functionality requires them.

[Figure 1–6](#_bookmark17) shows the multicycle custom instruction hardware port timing diagram.

**Figure 1–6. Multicycle Custom Instruction Timing Diagram**

clk

T0 T1 T2

T3 T4 T5 T6

clk\_en start

reset

dataa[] datab[]

valid

valid

done

result[]

valid

The processor asserts the active high start port on the first clock cycle of the custom instruction execution. At this time, the dataa and datab ports have valid values and remain valid throughout the duration of the custom instruction execution. The start signal is asserted for a single clock cycle.

For a fixed length multicycle custom instruction, after the instruction starts, the processor waits the specified number of clock cycles, and then reads the value on the result signal. For an *n*-cycle operation, the custom logic block must present valid data on the *n*th rising edge after the custom instruction begins execution.

For a variable length multicycle custom instruction, the processor waits until the active high done signal is asserted. The processor reads the result port on the same clock edge on which done is asserted. The custom logic block must present data on the result port on the same clock cycle on which it asserts the done signal.

The Nios II system clock feeds the custom logic block’s clk port, and the Nios II system’s master reset feeds the active high reset port. The reset port is asserted only when the whole Nios II system is reset.

The custom logic block must treat the active high clk\_en port as a conventional clock qualifier signal, ignoring clk while clk\_en is deasserted.

You can further optimize multicycle custom instructions by implementing the extended internal register file, or by creating external interface custom instructions. Refer to [“Extended Custom Instructions”](#_bookmark18), [“Internal Register File Custom](#_bookmark20) [Instructions” on page 1–9](#_bookmark20), or [“External Interface Custom Instructions” on page 1–10](#_bookmark23).

### Extended Custom Instructions

Extended custom instruction allows a single custom logic block to implement several different operations. Extended custom instructions use an index to specify which operation the logic block performs. The index can be as many as eight bits wide, allowing a single custom logic block to implement as many as 256 different operations.

[Figure 1–7](#_bookmark19) is a block diagram of an extended custom instruction with bit-swap, byte-swap, and half-word swap operations.

**Figure 1–7. Extended Custom Instruction with Swap Operations**

dataa[31..0]

bit-swap operation

byte-swap operation

half-word-swap operation

**Custom**

0

1

2

result[31..0]

n[1..0]

The custom instruction in [Figure 1–7](#_bookmark19) performs swap operations on data received at the dataa port. It uses the two-bit-wide n port to select the output from a multiplexer, determining which result is presented to the result port.

1 This logic is just a simple example, using a multiplexer on the output. You can implement function selection based on an index in any way that is appropriate for your application.

Extended custom instructions can be combinational or multicycle custom instructions. To implement an extended custom instruction, simply add an n port to your custom instruction logic. The bit width of the n port is a function of the number of operations the custom logic block can perform.

Extended custom instructions occupy multiple custom instruction indices. For example, the custom instruction illustrated in [Figure 1–7](#_bookmark19) occupies 4 indices, because n is two bits wide. Therefore, when this instruction is implemented in a Nios II system, 256 - 4 = 252 available indices remain.

For information about the custom instruction index, refer to [“Custom Instruction](#_bookmark35) [Assembly Software Interface” on page 2–3](#_bookmark35).

All extended custom instruction port operations are identical to those for the combinational and multicycle custom instructions, with the exception of the n port, which is not present in combinational and multicycle custom instructions. The n port timing is the same as that of the dataa port. For example, for an extended variable multicycle custom instruction, the processor presents the index value to the n port on the same rising edge of the clock at which start is asserted, and the n port remains stable during execution of the custom instruction.

### Internal Register File Custom Instructions

The Nios II processor allows custom instruction logic to access its own internal register file. This provides you the flexibility to specify if the custom instruction reads its operands from the Nios II processor’s register file or from the custom instruction’s own internal register file. In addition, a custom instruction can write its results to the local register file rather than to the Nios II processor’s register file.

Custom instructions containing internal register files use readra, readrb, and writerc signals to determine if the custom instruction should use the internal register file or the dataa, datab, and result signals. Ports a, b, and c specify the internal registers from which to read or to which to write. For example, if readra is deasserted (specifying a read operation from the internal register), the a signal value provides an index to the internal register file. Ports a, b, and c are five bits each, allowing you to address as many as 32 registers.

f For further details of Nios II custom instruction implementation, refer to the

[*Instruction Set Reference*](http://www.altera.com/literature/hb/nios2/n2cpu_nii51017.pdf)chapter of the *Nios II Processor Reference Handbook*.

[Table 1–4](#_bookmark21) lists the internal register file custom instruction-specific optional ports. Use the optional ports only if the custom instruction functionality requires them.

**Table 1–4. Internal Register File Custom Instruction Ports**

|  |  |  |  |
| --- | --- | --- | --- |
| **Port Name** | **Direction** | **Required** | **Description** |
| readra | Input | No | If readra is high, the Nios II processor supplies dataa;  if readra is low, custom instruction logic reads the internal register file indexed by a. |
| readrb | Input | No | If readrb is high, the Nios II processor supplies datab;  if readrb is low, custom instruction logic reads the internal register file indexed by b. |
| writerc | Input | No | If writerc is high, the Nios II processor writes to the result port; if writerc is low, custom instruction logic writes to the internal register file indexed by c. |
| a[4:0] | Input | No | Custom instruction internal register file index |
| b[4:0] | Input | No | Custom instruction internal register file index |
| c[4:0] | Input | No | Custom instruction internal register file index |

[Figure 1–8](#_bookmark22) shows a simple multiply-accumulate custom logic block.

**Figure 1–8. Multiply-accumulate Custom Logic Block**

Multiplier

D Q

CLR

Adder

dataa[31..0]

result[31..0]

datab[31..0]

writerc

When writerc is deasserted, the Nios II processor ignores the value driven on the result port. The accumulated value is stored in an internal register. Alternatively, the processor can read the value on the result port by asserting writerc. At the same time, the internal register is cleared so that it is ready for a new round of multiply and accumulate operations.

The readra, readrb, writerc, a, b, and c ports behave similarly to dataa. When the custom instruction begins, the processor presents the new values of the readra, readrb, writerc, a, b, and c ports on the rising edge of the processor clock. All six of these ports remain stable during execution of the custom instructions.

To determine how to handle the register file, custom instruction logic reads the active high readra, readrb, and writerc ports. The logic uses the a, b, and c ports as register file indexes. When readra or readrb is asserted, the custom instruction logic ignores the corresponding a or b port, and receives data from the dataa or datab port. When writerc is asserted, the custom instruction logic ignores the c port and writes to the result port.

All other custom instruction port operations behave the same as for combinational and multicycle custom instructions.

### External Interface Custom Instructions

Nios II custom instructions allow you to add an interface to communicate with logic outside of the processor’s datapath. At system generation, conduits propagate out to the top level of the SOPC Builder or Qsys system, where external logic can access the signals.

By enabling custom instruction logic to access memory external to the processor, external interface custom instructions extend the capabilities of the custom instruction logic.

[Figure 1–9](#_bookmark24) shows a multicycle custom instruction that has an external memory interface.

**Figure 1–9. Custom Instructions Allow the Addition of an External Interface**

Conduit Interface

dataa[31..0]

datab[31..0]

clk clk\_en

reset start

result[31..0]

done

Custom instruction logic can perform various tasks such as storing intermediate results or reading memory to control the custom instruction operation. The conduit interface also provides a dedicated path for data to flow into or out of the processor. For example, custom instruction logic with an external interface can feed data directly from the processor’s register file to an external first-in first-out (FIFO) memory buffer.

# 2. Software Interface

The Nios II custom instruction software interface abstracts logic implementation details from the application code. During the build process the Nios II software build tools generate macros that allow easy access from application code to custom instructions.

This chapter contains the following sections:

* [“Custom Instruction Examples” on page 2–1](#_bookmark27)
* [“Built-in Functions and User-defined Macros” on page 2–2](#_bookmark30)
* [“Custom Instruction Assembly Software Interface” on page 2–3](#_bookmark34)

## Custom Instruction Examples

[Example 2–1](#_bookmark28) shows a portion of the **system.h** header file that defines a macro for a bit-swap custom instruction. This bit-swap example accepts one 32-bit input and performs only one function.

**Example 2–1. Bit Swap Macro Definition**

#define ALT\_CI\_BITSWAP\_N 0x00

#define ALT\_CI\_BITSWAP(A) builtin\_custom\_ini(ALT\_CI\_BITSWAP\_N,(A))

In [Example 2–1](#_bookmark28), ALT\_CI\_BITWSWAP\_N is defined to be 0x0, which is the custom instruction’s index. The ALT\_CI\_BITSWAP(A) macro is mapped to a gcc built-in function that takes a single argument.

For more information about the gcc built-in functions, refer to [Appendix B, Custom](#_bookmark83) [Instruction Built-in Functions](#_bookmark83).

[Example 2–2](#_bookmark29) illustrates application code that uses the bit-swap custom instruction.

**Example 2–2. Bit Swap Instruction Usage**

#include "system.h"

int main (void)

{

int a = 0x12345678; int a\_swap = 0;

a\_swap = ALT\_CI\_BITSWAP(a); return 0;

}

The code in [Example 2–2](#_bookmark29) includes the **system.h** file to enable the application software to use the custom instruction macro definition. The example code declares two integers, a and a\_swap. Integer a is passed as input to the bit swap custom instruction and the results are loaded in a\_swap.

[Example 2–2](#_bookmark29) illustrates how most applications use custom instructions. The macros defined by the Nios II software build tools use C integer types only. Occasionally, applications require input types other than integers. In those cases, you can use a custom instruction macro to process non-integer return values.

1 You can define custom macros for Nios II custom instructions that allow other 32-bit input types to interface with custom instructions.

## Built-in Functions and User-defined Macros

The Nios II processor uses gcc built-in functions to map to custom instructions. By default, the integer type custom instruction is defined in a **system.h** file. However, by using built-in functions, software can use non-integer types with custom instructions. Fifty-two built-in functions are available to accommodate the different combinations of supported types.

Built-in function names have the following format:

builtin\_custom\_<*return type*>n<*parameter types*>

[Table 2–1](#_bookmark31) lists the 32-bit types supported by custom instructions as parameters and return types, as well as the abbreviations used in the built-in function names.

**Table 2–1. 32-bit Types Supported by Custom Instructions**

|  |  |
| --- | --- |
| **32-bit Type** | **Built-in Function Abbreviation** |
| int | i |
| float | f |
| void \* | p |

[Example 2–3](#_bookmark32) shows the prototype definitions for two built-in functions.

**Example 2–3. Two Example Built-in Function Prototypes**

void builtin\_custom\_nf (int n, float dataa); float builtin\_custom\_fnp (int n, void \* dataa);

The built-in function builtin\_custom\_nf takes a float as an input, and does not return a value. In contrast, the built-in function builtin\_custom\_fnp takes a pointer as input, and returns a float.

To support non-integer input types, define macros with mnemonic names that map to the specific built-in function required for the application.

f Refer to [Appendix B, Custom Instruction Built-in Functions](#_bookmark83) for detailed information and a list of built-in functions.

[Example 2–4](#_bookmark33) shows user-defined custom instruction macros used in an application.

**Example 2–4. Custom Instruction Macro Usage Example**

1. /\* define void udef\_macro1(float data); \*/
2. #define UDEF\_MACRO1\_N 0x00
3. #define UDEF\_MACRO1(A) builtin\_custom\_nf(UDEF\_MACRO1\_N, (A));
4. /\* define float udef\_macro2(void \*data); \*/
5. #define UDEF\_MACRO2\_N 0x01
6. #define UDEF\_MACRO2(B) builtin\_custom\_fnp(UDEF\_MACRO2\_N, (B)); 7.

8. int main (void) 9. {

10. float a = 1.789;

1. float b = 0.0;
2. float \*pt\_a = &a; 13.
3. UDEF\_MACRO1(a);
4. b = UDEF\_MACRO2((void \*)pt\_a);
5. return 0;

17. }

On lines 2 through 6, the user-defined macros are declared and mapped to the appropriate built-in functions. The macro UDEF\_MACRO1 takes a float as an input parameter and does not return anything. The macro UDEF\_MACRO2 takes a pointer as an input parameter and returns a float. Lines 14 and 15 show code that uses the two user-defined macros.

## Custom Instruction Assembly Software Interface

The Nios II custom instructions are also accessible in assembly code. This section describes the assembly interface.

Custom instructions are R-type instructions, containing:

* A 6-bit opcode
* Three 5-bit register index fields
* Three 1-bit fields for the readra, readrb, and writerc signals
* An 8-bit N field, used for the custom instruction index (opcode extension), and optionally including a function select subfield

[Figure 2–1](#_bookmark36) shows a diagram of the custom instruction word.

**Figure 2–1. Custom Instruction Format**

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

A

B

C

N

uP OPCode = Custom

readra readrb

writerc

**Instruction Fields:**

A = Register index of operand A B = Register index of operand B C = Register index of operand C

N = 8-bit number that selects instruction

readra = 1 if instruction uses processor’s register rA, 0 otherwise readrb = 1 if instruction uses processor’s register rB, 0 otherwise

writerc = 1 if instruction provides result for processor’s register rC, 0 otherwise

Bits 5–0 are the Nios II custom instruction opcode. The opcode for a custom instruction is 0x32.

f A list of opcodes appears in the *“Instruction Opcodes”* section in the [*Instruction Set*](http://www.altera.com/literature/hb/nios2/n2cpu_nii51017.pdf)[*Reference*](http://www.altera.com/literature/hb/nios2/n2cpu_nii51017.pdf)chapter of the *Nios II Processor Reference Handbook*.

The N field, bits 13:6, is the custom instruction index. The custom instruction index distinguishes between different custom instructions, allowing the Nios II processor to support as many as 256 distinct custom instructions. Depending on the type of custom instruction, the N field has one of the following functions:

* A unique custom instruction index, for logic that implements a single custom function
* An extended custom instruction index, for logic that implements several custom functions

[Example 2–5](#_bookmark37) shows the assembly language syntax for the custom instruction.

**Example 2–5. Custom Instruction Assembly Syntax**

custom N, xC, xA, xB

In the assembly code instruction in [Example 2–5](#_bookmark37), N is the custom instruction index, xC is the destination for the result[31:0] port, xA is the dataa port, and xB is the datab port. To access the Nios II processor’s register file, replace x with r. To access a custom register file, replace x with c. The use of r or c determines whether the custom instruction has the readra, readrb, and writerc bits held high or low. Refer to

[Figure 2–1](#_bookmark36) for the location of these three bits in the custom instruction low-level format.

[Example 2–6](#_bookmark38), [Example 2–7](#_bookmark39), and [Example 2–8](#_bookmark40) demonstrate the syntax for custom instruction assembler calls.

**Example 2–6. Assembly Language Call to Customer Instruction I**

custom 0, r6, r7, r8

[Example 2–6](#_bookmark38) shows a call to a custom instruction with index 0. The input to the instruction is the current contents of the Nios II processor registers r7 and r8, and the results are stored in the Nios II processor register r6.

**Example 2–7. Assembly Language Call to Customer Instruction II**

custom 3, c1, r2, c4

[Example 2–7](#_bookmark39) shows a call to a custom instruction with index 3. The input to the instruction is the current contents of the Nios II processor register r2 and the custom register c4, and the results are stored in custom register c1.

**Example 2–8. Assembly Language Call to Customer Instruction III**

custom 4, r6, c9, r2

[Example 2–8](#_bookmark40) shows a call to a custom instruction with index 4. The input to the instruction is the current contents of the custom register c9 and the Nios II processor register r2, and the results are stored in Nios II processor register r6.

f For more information about the binary format of custom instructions, refer to the

[*Instruction Set Reference*](http://www.altera.com/literature/hb/nios2/n2cpu_nii51017.pdf)chapter of the *Nios II Processor Reference Handbook*.

# 3. Implementing a Nios II Custom

**Instruction in Qsys**

This chapter describes the process of implementing a Nios II custom instruction with the Qsys component editor. The component editor enables you to create new Qsys components, including Nios II custom instructions. This chapter also describes the process of accessing Nios II custom instructions from software.

For detailed information about the Qsys component editor, refer to [*Creating Qsys*](http://www.altera.com/literature/hb/qts/qsys_components.pdf)[*Components*](http://www.altera.com/literature/hb/qts/qsys_components.pdf)in volume 1 of the *Quartus II Handbook*.

## Setting up the System

Before continuing with the lab, choose the board that you are going to use as your target. It is recommended that you choose a board you have not used before to become more comfortable with all of the boards. Also choose a location to serve as your project directory (<project\_directory>); ideally this is an easily accessible location associated with Lab3.

NOTE: Your project directory path should NOT have any spaces. If it does, this project will not work. Make your project in a location such as *C:\Users\<your UVA ID>\FPGA\_Projects\Lab3*

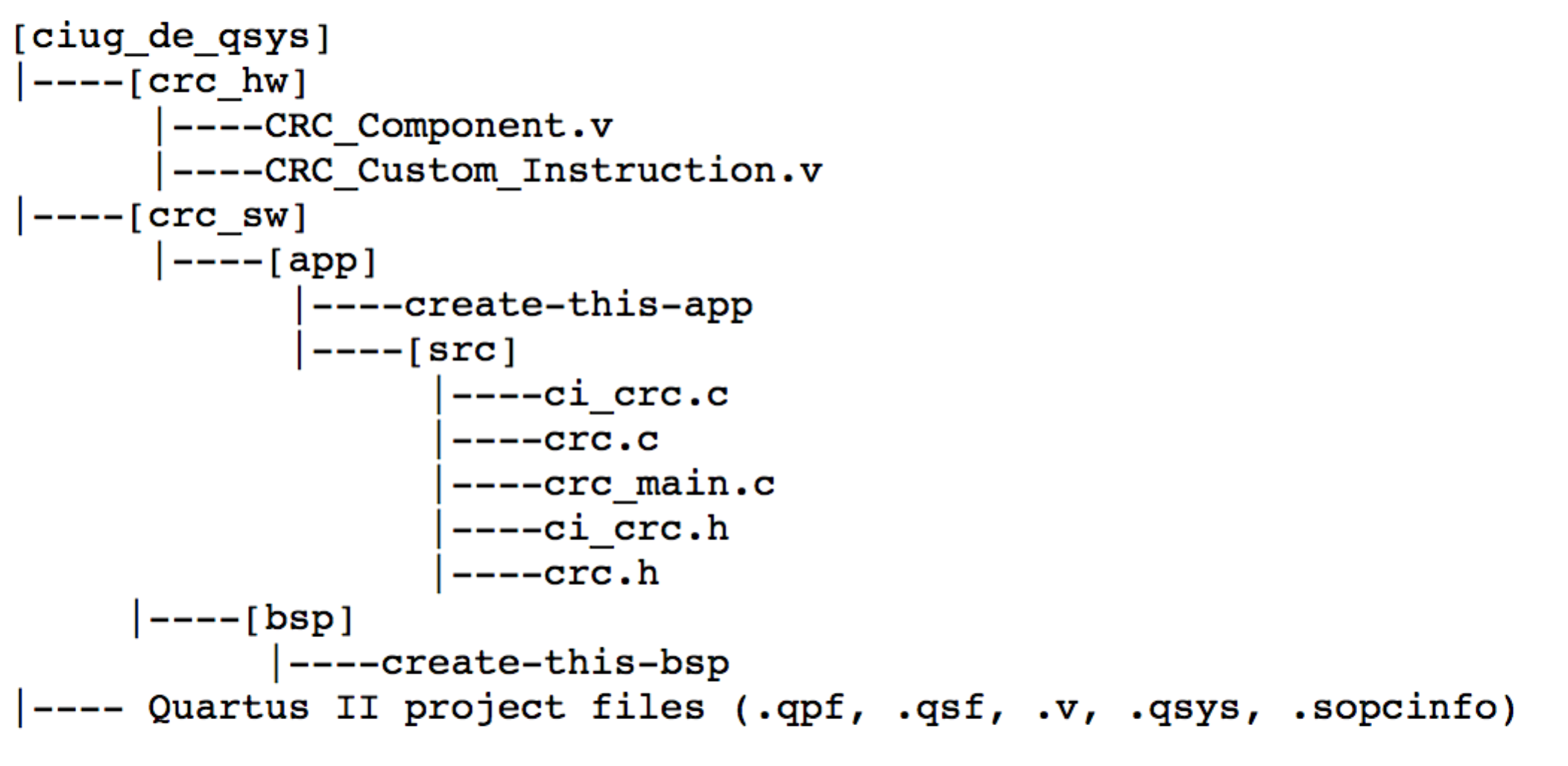
1. Copy the "golden" version of the SOPC directory to your own project directory (copy the whole directory) for your development board provided by Terasic on the (downloaded) CDROMs for the boards. These are found in the CD directory in <board\_name>\_demonstrations:

.../DE0-Nano\_SOPC\_DEMO/ (for DE0-nano) .../DE0\_NIOS\_SDCARD/ (for DE0) .../DE2\_115\_golden\_sopc/ (for DE2-115)

2. Inside the copied directory rename the main <board\_name>.qpf to <board\_name>\_Lab3.qpf (i.e. append \_Lab3 to the name), where <board\_name>.qpf is:

.../DE0-Nano\_SOPC\_DEMO/DE0\_Nano.qpf (for DE0-nano) .../DE0\_NIOS\_SDCARD/DE0.qpf (for DE0) .../DE2\_115\_golden\_sopc/de2\_115\_golden.sopc.qpf (for DE2-115)

3. Download the attached ciug\_de\_qsys.zip file and extract the contents. Copy the crc\_hw and crc\_sw directories to your <project\_directory>. The directory should look like this:



4. Start QUARTUS II by double clocking the renamed .qpf file.

5. From within QUARTUS II, start QSYS (Tools->QSYS). QSYS will ask to open a QSYS file. Depending on your board, open the .sopc file or .qsys file and let the tool update the system if necessary.

6. In Qsys disable (by unchecking the check boxes associated with the blocks, not by removing) some of the blocks that are not necessary. Make sure you keep anything related to the clocks, CPU, JTAG, PLLs, Timer, SDRAM, and anything else required for normal operation. Verify that you aren’t missing anything by checking the message board in the lower part of the screen for error message. The system should have no errors before being **generated**.

7. In QUARTUS II, you now need to edit the main ‘wrapper file’ (go to Hierarchy view, and open the top level file) for the system, in Verilog, in order to disable the ports for the blocks that you removed in the previous step. Do not delete anything; just add comment characters in front of the port names and the associated Verilog. This process will be iterative. Add some comments, try to compile, look for errors, add more comments, etc, until you end up with an error-free compilation.  
\*\*\* Watch out for the warnings!

8. At this point, jump ahead and complete the custom instruction tutorial below to create a custom CRC instruction; once complete, return to 9.

[Click here to jump to the CRC instruction tutorial.](#Design_Example:_Cyclic_Redundancy_Check)

9. Generate the system by selecting Generate -> Generate. Leave the default settings, and start synthesizing the system. Once successful, close out of QSYS.

Since you are using a PLL, make sure you add timing rules. Go to **Tools->TimeQuest Timing Analyzer**, create a new SDC file (under File-> New), and add the lines:

*create\_clock -period 20.000 -name CLOCK\_50 CLOCK\_50  
derive\_pll\_clocks  
derive\_clock\_uncertainty*

Save this file by overwriting the existing SDC file in the main project directory.

10. The next step is to set up the system for compilation. This will compile all of the Verilog into a system configuration. To do this, open QUARTUS II, and select the FILES tab under the PROJECT NAVIGATOR in the top-left corner. Right-click FILES and select “ADD/REMOVE FILES IN PROJECT…”.

11. Click the ellipses next to file name in the Settings window, and navigate to the Synthesis directory. This should be located in your Project Directory in the folder called <board\_name>. In the synthesis directory select the .QIP and .V file and add them.

12. At this point, compile the system by selecting Processing -> Start Compilation.

**Note: You may run into errors; remove the Verilog files that the errors refer to. You will need to remove the audio and vga Verilog files.**

13. The next step is to download the bitstream onto the FPGA and run the example software. This program will compute the CRC using the general purpose CPU in software, and compare that result against using the custom instruction you added in the previous steps. Open the NIOS II Command Shell, and change directory to <project\_directory>. (cd <project\_directory>)

14. Configure the FPGA by running the following command:

**Note: This bitstream file must be the time\_limited version**

nios2-configure-sof <board\_name>\_Lab3\_time\_limited.sof

\*Note: Because you are using the time\_limited version, the pc will need to remain connected to the board at all times. Let this program continue running; do not cancel.

15. Open another NIOS II Command Shell window, and change to the <project\_directory> directory. Now change directory to <project\_directory>/crc\_sw/app.

16. Run the following command for building the application software:

./create-this-app

NOTE: Make sure that the *crc\_sw* folder is in the main project directory and NOT a subdirectory, or else you will get errors!

1. When the command completes, run the application with the command:

nios2-download –g crc.elf && nios2-terminal

## Design Example: Cyclic Redundancy Check

The cyclic redundancy check (CRC) algorithm detects the corruption of data during transmission. It detects a higher percentage of errors than a simple checksum. The CRC calculation consists of an iterative algorithm involving XOR and shift operations. These operations are carried out concurrently in hardware and iteratively in software. Because the operations are carried out concurrently, the execution is much faster in hardware.

The CRC design files demonstrate the steps to implement an extended multicycle Nios II custom instruction. These design files are available for you to download from the class wiki site.

## Implementing Custom Instruction Hardware in Qsys

This section describes the custom instruction tool-flow, and walks you through the process of implementing a Nios II custom instruction. Implementing a Nios II custom instruction hardware entails the following tasks:

* [“Opening the Component Editor”](#_bookmark64)
* [“Adding the HDL Files”](#_bookmark65)
* [“Configuring the Custom Instruction Signal Type” on page 4–3](#_bookmark66)
* [“Setting Up the Custom Instruction Interfaces” on page 4–3](#_bookmark67)
* [“Setting the Details” on page 4–4](#_bookmark68)
* [“Saving and Adding the Custom Instruction” on page 4–5](#_bookmark69)
* [“Generating the System and Compiling in the Quartus II Software” on page 4–5](#_bookmark70)

The following sections detail the steps required to set up the design example environment and to perform the list of tasks.

### Setting up the Design Environment for the Design Example

NOTE: Greyed-out text is deprecated; you can skip it.

Before you start the design example, you must set up the design environment to accommodate the processes described in the following sections. To set up the design example environment, follow these steps:

1. Open the **ciug\_de\_qsys.zip** file from the class Collab site and extract all the files to a new directory.
2. Follow the instructions in the Quartus II Project Setup section in the **readme\_qsys.txt** file in the extracted design files. The instructions direct you to determine a <*project\_dir*> working directory for the project and to open the design example project in the Quartus II software.

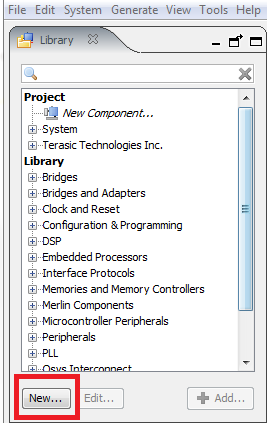
### Opening the Component Editor

After you follow the steps in the Quartus II Project Setup section of the **readme\_qsys.txt** file, you have a Quartus II project located in the <*project\_dir*> directory and open in the Quartus II software.

To open the component editor, follow these steps:

1. To open Qsys, on the Tools menu, click **Qsys**.
2. In Qsys, on the File menu, click **Open**.
3. Browse to the <*project\_dir*> directory if necessary, select the **.qsys** file, and click

#### Open.

1. On the Qsys **Component Library** tab, click **New**. The component editor appears, displaying the **Introduction** tab.  
   

### Adding the HDL Files

To specify the synthesis HDL files for your custom instruction, follow these steps:  
(NOTE: the Qsys interface has changed significantly throughout the years, therefore not all buttons will be there. Use your instincts as an engineer to debug any issues you run into.)

1. Under **Name** and **Display Name**, replace “new\_component” with “CRC,” without the quotation marks.
2. Click **Next** to display the **HDL Files** tab.
3. Click **Add**. Under “Synthesis Files” click the “+” button
4. Browse to <*project\_dir*>/**crc\_hw**, the location of the HDL files for this design example. The files were downloaded from the wiki in *Ciug\_de\_sys.zip*.
5. Select the **CRC\_Custom\_Instruction.v** and **CRC\_Component.v** files and click

#### Open.

#### Click on “Analyze Synthesis Files”…

1. Ensure the **Synth** setting is turned on for both files. This setting directs the component editor to generate models for synthesis. (NOTE: This setting has been deprecated).
2. Turn on the **Top** parameter for the **CRC\_Custom\_Instruction.v** file, to indicate it is the top-level HDL file for this custom instruction. Double-click on “no attributes”.
3. The Quartus II Analysis and Synthesis program checks the design for errors when you change the **Top** setting. Confirm that no error message appears.
4. Click “Analyze Synthesis Files” again. At this point you may encounter errors; **it is safe to ignore these for now.**
5. Click **Top Level Module** and select the name of the top-level module of your custom instruction logic. The top-level module for this design example is **CRC\_Custom\_Instruction**.
6. To enable simulating the system in the ModelSim simulator, you can turn on **Sim** for both files. In cases other than the design example, you may need to add additional simulation files in the component editor **HDL Files** tab, turn on **Sim** and turn off **Synth** for these simulation-only files, and turn off **Sim** for the synthesis-only files.

### Configuring the Custom Instruction Signal Type

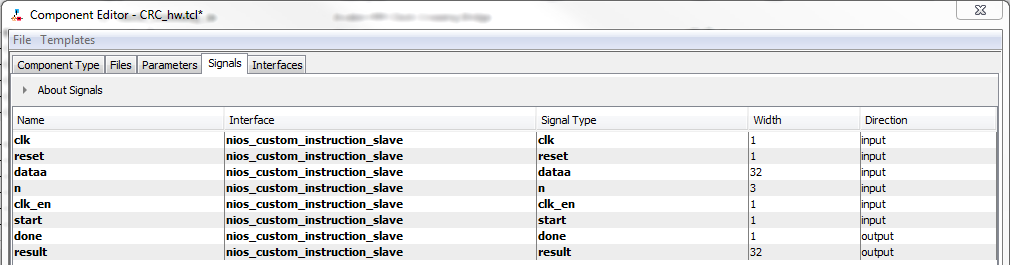
To configure the custom instruction signal type, follow these steps:

1. Click **Next** twiceto display the **Signals** tab.
2. For each port listed in the tab, follow these steps:
   1. Select the port.
   2. In the **Interface** column, select the name of the interface to which you want to assign the port (i.e. signal name).

For the design example, for the first port (clk) you configure, select **New Custom Instruction Slave (will change itself to nios\_custom\_instruction\_slave)**. For the remaining ports, select the name created for the first port, which is **nios\_custom\_instruction\_slave**. These selections ensure that the ports appear together on a single interface, using the name assumed by the design example **crc\_hw** files.

* 1. In the **Signal Type** column, select the appropriate signal type for the port. The available signal types are described in [“Custom Instruction Types” on](#_bookmark7)

[page 1–3](#_bookmark7). In the design example, the correct signal type for each signal has the same name as the signal.



### Setting Up the Custom Instruction Interfaces

To set up the custom instruction interfaces, follow these steps:

1. Click **Next** to display the **Interfaces** tab.
2. If the **Remove Interfaces With No Signals** button is active, click it.
3. Ensure that a single interface remains, with **Name** set to the name in the Signals tab. For the design example, maintain the interface name **nios\_custom\_instruction\_slave.**

1 If you rename an interface by changing the value in the **Name** field, the **Signals** tab **Interface** column value changes automatically. The value shown in the block diagram updates when you change tabs and return to the **Interfaces** tab.

1. Ensure the **Type** for this interface is **Custom Instruction Slave**.
2. For **Clock Cycles**, type 0 for a variable multicycle type custom instruction, and otherwise type the number of clock cycles your custom instruction logic requires. The design example builds a variable multicycle type custom instruction.

1 If the interface includes a done signal and a clk signal, the component editor automatically infers that the interface is a variable multicycle type custom instruction interface, and sets the value to 0.

1. For **Clock Cycle Type**, type Variable for a variable multicycle type custom instruction, Multicycle for a fixed multicycle type custom instruction, or Combinatorial for a combinational type custom instruction. Because the design example builds a variable multicycle type custom instruction, **Clock Cycle Type** is set to **Variable**.

1 If the interface does not include a clk signal, the component editor automatically infers that the interface is a combinational type interface. If the interface includes a clk signal, the component editor automatically infers that the interface is a multicycle interface. If the interface does not include a done signal, the component editor infers that the interface is a fixed multicycle type interface. If the interface includes a done signal, the component editor infers that the interface is a variable multicycle type interface.

1. For **Operands**, type the number of operands for your custom instruction. For the design example, type 1.

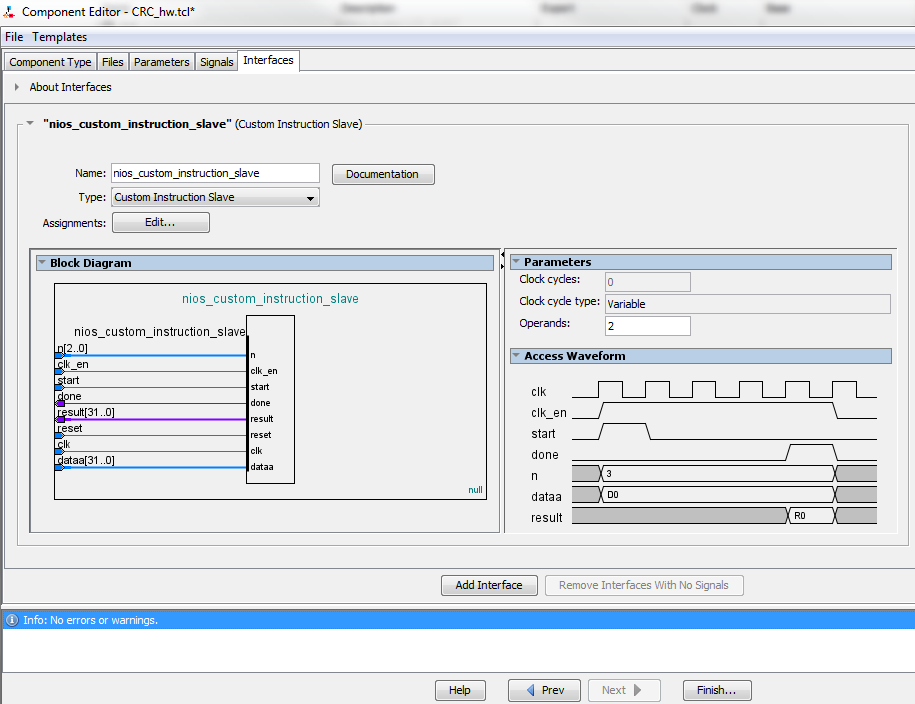
If your custom instruction logic requires additional interfaces, either to the Avalon-MM fabric or outside the Qsys system, you can specify the additional interfaces in the **Interfaces** tab. The design example does not require additional interfaces.

1 Most custom instructions use some combination of standard custom instruction ports, such as dataa, datab, and result, and do not require additional interfaces.

The following instructions provide the information you need if a custom instruction in your own design requires additional interfaces. If you are walking through the CRC design example, proceed to [“Setting the Details”](#_bookmark68).

To specify additional interfaces on the **Interfaces** tab, follow these steps:

1. Click **Add Interface**. The new interface has **Custom Instruction Slave** interface type by default.
2. Click “Remove Interfaces With No Signals” if there are lots of interfaces



1. For **Type**, select the desired interface type.
2. Set the parameters for the newly created interface according to your system requirements.

### Setting the Details

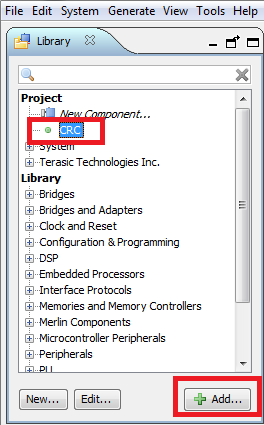
NOTE: This section has been deprecated.

To specify the final details in the custom instruction parameter editor, follow these steps:

1. Click **Next** to display the **HDL Parameters** tab.
2. Click **Next** to display the **Library Info** tab.
3. For **Name** and for **Display Name**, type CRC.
4. For **Version**, type 1.0.
5. Leave the **Group** field blank.
6. Optionally, set the **Description**, **Created by**, and **Icon** fields as you prefer.
7. If the bottom pane of the dialog box displays something other than the message **Info: No errors or warnings**, review the other tabs to ensure you followed the instructions correctly.

### Saving and Adding the Custom Instruction

To save the custom instruction and add it to your Nios II processor, follow these steps:

1. Click **Finish**. A dialog box prompts you to save your changes before exiting.
2. Click **Yes, Save**. The new custom instruction appears in the Qsys Component Library.
3. In the Qsys Component Library, under Library, select **CRC**, the new custom instruction you created in the design example.
4. Click **Add** to add the new instruction to the Qsys system.
5. In the **Connections** panel, connect the new **CRC\_0** component’s **nios\_custom\_instruction\_slave** interface to the **cpu** component’s **custom\_instruction\_master** interface.  
   

### Generating the System and Compiling in the Quartus II Software

After you add the custom instruction logic to the system, you can generate the Qsys system and compile it in the Quartus II software.

To generate the system and compile, follow these steps:

1. In Qsys, on the **Generation** tab, turn on **Create HDL design files for synthesis**.
2. Click **Generate**. System generation may take several seconds to complete.
3. After system generation completes, on the File menu, click **Exit**.
4. In the Quartus II software, under the **Project menu tab**, click **Add/Remove Files in Project**.
5. Ensure that the **.qip** file in the **synthesis** subdirectory (under *<board name>\_qsys* or *<board name>\_sopc* ) is added to the project.
6. On the Processing menu, click **Start Compilation**.

f For detailed information about Qsys systems and their generation, refer to the [*System*](http://www.altera.com/literature/hb/qts/qsys_section.pdf)[*Design with Qsys*](http://www.altera.com/literature/hb/qts/qsys_section.pdf)section in volume 1 of the *Quartus II Handbook*.

Now return to step 9.

## Accessing the Custom Instruction from Software

Adding a custom instruction to a Nios II processor results in a significant change to the Qsys system. In this section, you create and build a new software project using the Nios II software build flow, and run the software that accesses the custom instruction. The software source files are included in the downloadable design files.

[Table 3–1](#_bookmark72) lists the CRC application software source files and their corresponding descriptions.

**Table 3–1. CRC Application Software Source Files**

|  |  |
| --- | --- |
| **File Name** | **Description** |
| **crc\_main.c** | Main program that populates random test data, executes the CRC both in software and with the custom instruction, validates the output, and reports the processing time. |
| **crc.c** | Software CRC algorithm run by the Nios II processor. |
| **crc.h** | Header file for **crc.c**. |
| **ci\_crc.c** | Program that accesses CRC custom instruction. |
| **ci\_crc.h** | Header file for **ci\_crc.c**. |

To run the application software, you must create an Executable and Linking Format File (**.elf**) first. To create the **.elf** file, follow the instructions in the Nios II Software Build Flow section in the **readme\_qsys.txt** file in the extracted design files.

The application program runs three implementations of the CRC algorithm on the same pseudo-random input data: an unoptimized software implementation, an optimized software implementation, and the custom instruction CRC. The program calculates the processing time and throughput for each of the versions, to demonstrate the improved efficiency of a custom instruction compared to a software implementation.

[Example 3–1](#_bookmark73) shows the output from the application program run on a Nios II Embedded Evaluation Kit.

The output shows that the custom instruction CRC is more than 70 times faster than the unoptimized CRC calculated purely in software and is more than 45 times faster than the optimized version of the software CRC. The results you see using different target device and board may vary depending on the memory characteristics of the board and the clock speed of the device, but these ratios are representative.

**Example 3–1. Output of the CRC Design Example Software Run**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Comparison between software and custom instruction CRC32

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

System specification

--------------------

System clock speed = 62.5 MHz Number of buffer locations = 16 Size of each buffer = 65535 bytes

Initializing all of the buffers with pseudo-random data

-------------------------------------------------------

Initialization completed

Running the software CRC

------------------------

Completed

Running the optimized software CRC

----------------------------------

Completed

Running the custom instruction CRC

----------------------------------

Completed

Validating the CRC results from all implementations

---------------------------------------------------

All CRC implementations produced the same results

Processing time for each implementation

---------------------------------------

Software CRC = 9928.41 ms

Optimized software CRC = 6414.63 ms Custom instruction CRC = 137.54 ms

Processing throughput for each implementation

---------------------------------------------

Software CRC = 0.84 Mbps

Optimized software CRC = 1.31 Mbps Custom instruction CRC = 60.99 Mbps

Speedup ratio

-------------

Custom instruction CRC vs software CRC = 72.2

Custom instruction CRC vs optimized software CRC = 46.6 Optimized software CRC vs software CRC = 1.5

## 

## Using the User-defined Custom Instruction Macro

The design example software uses a user-defined macro to access the CRC custom instruction. [Example 3–2](#_bookmark75) shows the macro that is defined in the **ci\_crc.c** file.

**Example 3–2. CRC Custom Instruction Macro Usage**

#define CRC\_CI\_MACRO(n, A)

\_\_builtin\_custom\_ini(ALT\_CI\_CRC\_0\_N + (n & 0x7), (A))

This macro takes a single int type input operand and returns an int type value. The CRC custom instruction has extended type; the n value in the macro CRC\_CI\_MACRO() indicates the operation to be performed by the custom instruction. The custom instruction index is added to the value of n. The n value is masked because the n port of a custom instruction has only three bits.

To initialize the custom instruction, for example, you can add the initialization code in [Example 3–3](#_bookmark76) to your application software.

**Example 3–3. Using the User-defined Macro to Initialize the Custom Instruction Logic**

/\* Initialize the custom instruction CRC to the initial remainder value: \*/ CRC\_CI\_MACRO (0,0);

For details of each operation of the CRC custom instruction and the corresponding index value n, refer to the comments in the **ci\_crc.c** file.

[Example 4–2](#_bookmark75) and [Example 4–3](#_bookmark76) demonstrate that you can define the macro in your application to accommodate your requirements. For example, you can determine the number and type of input operands, decide whether to assign a return value, and vary the custom instruction index. However, the macro definition and usage must be consistent with the port declarations of the custom instruction. For example, if you define the macro to return an int value, the custom instruction must have a result port.

f For details about writing software for Nios II custom instructions, refer to [Chapter 2,](#_bookmark26) [Software Interface](#_bookmark26).

# Custom Instruction Templates

This appendix provides VHDL and Verilog HDL custom instruction wrapper file templates that you can reference when writing custom instructions in VHDL and Verilog HDL.

f Full sets of template files are available in the <*nios2eds installation directory*>**/examples/verilog/custom\_instruction\_templates** and <*nios2eds installation directory*>**/examples/vhdl/custom\_instruction\_templates** directories.

## VHDL Custom Instruction Template

[Example A–1](#_bookmark79) shows a VHDL custom instruction template for an internal register type custom instruction.

**Example A–1. VHDL Custom Instruction Template for Internal Register Logic Type Instruction**

-- VHDL Custom Instruction Template File for Internal Register Logic

library ieee;

use ieee.std\_logic\_1164.all;

entity custominstruction is port(

signal clk: in std\_logic;

-- CPU system clock (required for multicycle or extended multicycle) signal reset: in std\_logic;

-- CPU master asynchronous active high reset

-- (required for multicycle or extended multicycle) signal clk\_en: in std\_logic;

-- Clock-qualifier (required for multicycle or extended multicycle) signal start: in std\_logic;

-- Active high signal used to specify that inputs are valid

-- (required for multicycle or extended multicycle) signal done: out std\_logic;

-- Active high signal used to notify the CPU that result is valid

-- (required for variable multicycle or extended variable multicycle) signal n: in std\_logic\_vector(7 downto 0);

-- N-field selector (required for extended);

-- Modify width to match the number of unique operations in the instruction signal dataa: in std\_logic\_vector(31 downto 0);-- Operand A (always required) signal datab: in std\_logic\_vector(31 downto 0);-- Operand B (optional)

signal a: in std\_logic\_vector(4 downto 0);-- Internal operand A index register signal b: in std\_logic\_vector(4 downto 0);-- Internal operand B index register signal c: in std\_logic\_vector(4 downto 0);-- Internal result index register signal readra: in std\_logic;

-- Read operand A from CPU (otherwise use internal operand A) signal readrb: in std\_logic;

-- Read operand B from CPU (otherwise use internal operand B) signal writerc: in std\_logic;

-- Write result to CPU (otherwise write to internal result)

signal result: out std\_logic\_vector(31 downto 0)-- result (always required)

);

end entity custominstruction;

architecture a\_custominstruction of custominstruction is

-- local custom instruction signals begin

-- custom instruction logic (note: external interfaces can be used as well)

-- Use the n[7..0] port as a select signal on a multiplexer

-- to select the value to feed result[31..0]

end architecture a\_custominstruction;

**Appendix A: Custom Instruction Templates A–3**

Verilog HDL Custom Instruction Template Example

## Verilog HDL Custom Instruction Template Example

[Example A–2](#_bookmark81) shows a Verilog HDL custom instruction template for an internal register type custom instruction.

**Example A–2. Verilog HDL Custom Instruction Template for Internal Register Logic Type Instruction**

// Verilog Custom Instruction Template File for Internal Register Logic

module custominstruction(

clk, // CPU system clock (required for multicycle or extended multicycle) reset, // CPU master asynchronous active high reset

// (required for multicycle or extended multicycle)

clk\_en,// Clock-qualifier (required for multicycle or extended multicycle) start, // Active high signal used to specify that inputs are valid

// (required for multicycle or extended multicycle)

done, // Active high signal used to notify the CPU that result is valid

// (required for variable multicycle or extended variable multicycle) n, // N-field selector (required for extended)

dataa, // Operand A (always required) datab, // Operand B (optional)

a, // Internal operand A index register b, // Internal operand B index register c, // Internal result index register

readra,// Read operand A from CPU (otherwise use internal operand A) readrb, // Read operand B from CPU (otherwise use internal operand B) writerc,// Write result to CPU (otherwise write to internal result) result // Result (always required)

);

//INPUTS

inputclk; inputreset; inputclk\_en; inputstart;

input[7:0]n;// modify width to match the number of unique operations in the instruction input[4:0]a;

input[4:0]b;

input[4:0]c; inputreadra; inputreadrb; inputwriterc; input[31:0]dataa; input[31:0]datab;

//OUTPUTS

outputdone; output[31:0]result;

// custom instruction logic (note: external interfaces can be used as well)

// Use the n[7..0] port as a select signal on a multiplexer

// to select the value to feed result[31..0]

endmodule

# Custom Instruction Built-in Functions

The Nios II gcc compiler, nios2-elf-gcc, is customized with built-in functions to support custom instructions. This section lists the built-in functions.

f For more information about gcc built-in functions, refer to [**www.gnu.org**.](http://www.gnu.org/)

Nios II custom instruction built-in functions have the following return types:

* void
* int
* float
* pointer

## Built-in Functions that Return a Void Value

The following built-in functions return a void value:

* void builtin\_custom\_n (int n);
* void builtin\_custom\_ni (int n, int dataa);
* void builtin\_custom\_nf (int n, float dataa);
* void builtin\_custom\_np (int n, void \*dataa);
* void builtin\_custom\_nii (int n, int dataa, int datab);
* void builtin\_custom\_nif (int n, int dataa, float datab);
* void builtin\_custom\_nip (int n, int dataa, void \*datab);
* void builtin\_custom\_nfi (int n, float dataa, int datab);
* void builtin\_custom\_nff (int n, float dataa, float datab);
* void builtin\_custom\_nfp (int n, float dataa, void \*datab);
* void builtin\_custom\_npi (int n, void \*dataa, int datab);
* void builtin\_custom\_npf (int n, void \*dataa, float datab);
* void builtin\_custom\_npp (int n, void \*dataa, void \*datab);

## Built-in Functions that Return a Value of Type Int

The following built-in functions return a value of type int:

* int builtin\_custom\_in (int n);
* int builtin\_custom\_ini (int n, int dataa);
* int builtin\_custom\_inf (int n, float dataa);
* int builtin\_custom\_inp (int n, void \*dataa);
* int builtin\_custom\_inii (int n, int dataa, int datab);
* int builtin\_custom\_inif (int n, int dataa, float datab);
* int builtin\_custom\_inip (int n, int dataa, void \*datab);
* int builtin\_custom\_infi (int n, float dataa, int datab);
* int builtin\_custom\_inff (int n, float dataa, float datab);
* int builtin\_custom\_infp (int n, float dataa, void \*datab);
* int builtin\_custom\_inpi (int n, void \*dataa, int datab);
* int builtin\_custom\_inpf (int n, void \*dataa, float datab);
* int builtin\_custom\_inpp (int n, void \*dataa, void \*datab);

## Built-in Functions that Return a Value of Type Float

The following built-in functions return a value of type float:

* float builtin\_custom\_fn (int n);
* float builtin\_custom\_fni (int n, int dataa);
* float builtin\_custom\_fnf (int n, float dataa);
* float builtin\_custom\_fnp (int n, void \*dataa);
* float builtin\_custom\_fnii (int n, int dataa, int datab);
* float builtin\_custom\_fnif (int n, int dataa, float datab);
* float builtin\_custom\_fnip (int n, int dataa, void \*datab);
* float builtin\_custom\_fnfi (int n, float dataa, int datab);
* float builtin\_custom\_fnff (int n, float dataa, float datab);
* float builtin\_custom\_fnfp (int n, float dataa, void \*datab);
* float builtin\_custom\_fnpi (int n, void \*dataa, int datab);
* float builtin\_custom\_fnpf (int n, void \*dataa, float datab);
* float builtin\_custom\_fnpp (int n, void \*dataa, void \*datab);

## Built-in Functions that Return a Pointer Value

The following built-in functions return a pointer value:

* void \* builtin\_custom\_pn (int n);
* void \* builtin\_custom\_pni (int n, int dataa);
* void \* builtin\_custom\_pnf (int n, float dataa);
* void \* builtin\_custom\_pnp (int n, void \*dataa);
* void \* builtin\_custom\_pnii (int n, int dataa, int datab);
* void \* builtin\_custom\_pnif (int n, int dataa, float datab);
* void \* builtin\_custom\_pnip (int n, int dataa, void \*datab);

**Appendix B: Custom Instruction Built-in Functions B–3**

Built-in Functions that Return a Pointer Value

* void \* builtin\_custom\_pnfi (int n, float dataa, int datab);
* void \* builtin\_custom\_pnff (int n, float dataa, float datab);
* void \* builtin\_custom\_pnfp (int n, float dataa, void \*datab);
* void \* builtin\_custom\_pnpi (int n, void \*dataa, int datab);
* void \* builtin\_custom\_pnpf (int n, void \*dataa, float datab);
* void \* builtin\_custom\_pnpp (int n, void \*dataa, void \*datab);

If you add a floating point custom instruction to your system, conflicts could arise in the software build process. This appendix describes how to handle these conflicts.

Both Qsys and SOPC Builder offer a predefined **Floating Point Hardware** component that you can add to your system. When you add this component to your system, a parameter editor displays, and you can turn on an option to include a floating point divider. The parameter editor notifies you that the component generates

single-precision floating point hardware.

When you add the floating point custom instruction to your system and run the software build process, flags are added to your nios2-elf-gcc command line. These flags specify the opcode extensions of the custom instructions that are called by the software and select the appropriate library.

One of the following two flags appears:

* + -mcustom-fpu-cfg=60-1
  + -mcustom-fpu-cfg=60-2

The flag value 60-1 indicates your system does not include a custom instruction floating point divider, and the flag value 60-2 indicates your system includes a custom instruction floating point divider.

The -mcustom-fpu-cfg flags forces the use of single-precision constants. To allow double-precision constants, you must remove the -mcustom-fpu-cfg flag and replace it with different individual compiler flags, depending on the flag value.

To enable double-precision floating point constants, follow one of these steps:

* + Replace the command-line option string -mcustom-fpu-cfg=60-1 with the following command-line option string:

-mcustom-fmuls=252,-mcustom-fadds=253,-mcustom-fsubs=254

* + Replace the command-line option string -mcustom-fpu-cfg=60-2 with the following command-line option string:

-mcustom-fmuls=252,-mcustom-fadds=253,-mcustom-fsubs=254,-mcustom-fdivs=255

1 Change the flags only if required. Replacing the -mcustom-fpu-cfg flag disables floating point custom instruction support in your library calls, forcing the system to use the emulated or slower version of the instruction instead.